[What is Ownership? 1](#__RefHeading___Toc4829_1947540165)

[Ownership Rules 3](#__RefHeading___Toc4831_1947540165)

[Variable Scope 3](#__RefHeading___Toc4833_1947540165)

[The String Type 4](#__RefHeading___Toc4835_1947540165)

[Memory and Allocation 5](#__RefHeading___Toc4837_1947540165)

[Ways Variables and Data Interact: Move 6](#__RefHeading___Toc4839_1947540165)

[Ways Variables and Data Interact: Clone 16](#__RefHeading___Toc4841_1947540165)

[Stack-only Data: Copy 17](#__RefHeading___Toc4843_1947540165)

[Ownership and Functions 18](#__RefHeading___Toc4845_1947540165)

[Return Values and Scope 19](#__RefHeading___Toc4847_1947540165)

[References and Borrowing 20](#__RefHeading___Toc4849_1947540165)

[Mutable References 23](#__RefHeading___Toc4851_1947540165)

[Dangling References 25](#__RefHeading___Toc4853_1947540165)

[The Rules of References 26](#__RefHeading___Toc4855_1947540165)

[Slices 27](#__RefHeading___Toc4857_1947540165)

[String Slices 29](#__RefHeading___Toc4859_1947540165)

[String Literals are Slices 33](#__RefHeading___Toc4861_1947540165)

[String Slices as Arguments 33](#__RefHeading___Toc4863_1947540165)

[Other Slices 34](#__RefHeading___Toc4865_1947540165)

[Summary 34](#__RefHeading___Toc4867_1947540165)

Understanding Ownership

Ownership is Rust’s most unique feature, and it enables Rust to make memory safety guarantees without needing a garbage collector. Therefore, it’s important to understand how ownership works in Rust. In this chapter we’ll talk about ownership as well as several related features: borrowing, slices, and how Rust lays data out in memory.

What Is Ownership?

Rust’s central feature is ownership. Although the feature is straightforward to explain, it has deep implications for the rest of the language.

All programs have to manage the way they use a computer’s memory while running. Some languages have garbage collection that constantly looks for no longer used memory as the program runs; in other languages, the programmer must explicitly allocate and free the memory. Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. No run-time costs are incurred for any of the ownership features.

Because ownership is a new concept for many programmers, it does take some time to get used to. The good news is that the more experienced you become with Rust and the rules of the ownership system, the more you’ll be able to naturally develop code that is safe and efficient. Keep at it!

When you understand ownership, you’ll have a solid foundation for understanding the features that make Rust unique. In this chapter, you’ll learn ownership by working through some examples that focus on a very common data structure: strings.

PROD: START BOX

The Stack and the Heap

In many programming languages, we don’t have to think about the stack and the heap very often. But in a systems programming language like Rust, whether a value is on the stack or the heap has more of an effect on how the language behaves and why we have to make certain decisions. We’ll describe parts of ownership in relation to the stack and the heap later in this chapter, but here is a brief explanation.

Both the stack and the heap are parts of memory that is available to your code to use at runtime, but they are structured in different ways. The stack stores values in the order it gets them and removes the values in the opposite order. This is referred to as last in, first out. Think of a stack of plates: when you add more plates, you put them on top of the pile, and when you need a plate, you take one off the top. Adding or removing plates from the middle or bottom wouldn’t work as well! Adding data is called pushing onto the stack, and removing data is called popping off the stack.

The stack is fast because of the way it accesses the data: it never has to search for a place to put new data or a place to get data from because that place is always the top. Another property that makes the stack fast is that all data on the stack must take up a known, fixed size.

For data with a size unknown to us at compile time or a size that might change, we can store data on the heap instead. The heap is less organized: when we put data on the heap, we ask for some amount of space. The operating system finds an empty spot somewhere in the heap that is big enough, marks it as being in use, and returns to us a pointer to that location. This process is called allocating on the heap, and sometimes we just abbreviate the phrase as just “allocating.” Pushing values onto the stack is not considered allocating. Because the pointer is a known, fixed size, we can store the pointer on the stack, but when we want the actual data, we have to follow the pointer.

Think of being seated at a restaurant. When you enter, you state the number of people in your group, and the staff finds an empty table that fits everyone and leads you there. If someone in your group comes late, they can ask where you’ve been seated to find you.

Accessing data in the heap is slower than accessing data on the stack because we have to follow a pointer to get there. Contemporary processors are faster if they jump around less in memory. Continuing the analogy, consider a server at a restaurant taking orders from many tables. It’s most efficient to get all the orders at one table before moving on to the next table. Taking an order from table A, then an order from table B, then one from A again, and then one from B again would be a much slower process. By the same token, a processor can do its job better if it works on data that’s close to other data (as it is on the stack) rather than farther away (as it can be on the heap). Allocating a large amount of space on the heap can also take time.

When our code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function’s local variables get pushed onto the stack. When the function is over, those values get popped off the stack.

Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap so we don’t run out of space are all problems that ownership addresses. Once you understand ownership, you won’t need to think about the stack and the heap very often, but knowing that managing heap data is why ownership exists can help explain why it works the way it does.

PROD: END BOX

Ownership Rules

First, let’s take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate the rules:

Each value in Rust has a variable that’s called its owner.

There can only be one owner at a time.

When the owner goes out of scope, the value will be dropped.

Variable Scope

We’ve walked through an example of a Rust program already in Chapter 2. Now that we’re past basic syntax, we won’t include all the fn main() { code in examples, so if you’re following along, you’ll have to put the following examples inside a main function manually. As a result, our examples will be a bit more concise, letting us focus on the actual details rather than boilerplate code.

PROD: check xref for tutorial ch number

As a first example of ownership, we’ll look at the scopeof some variables. A scope is the range within a program for which an item is valid. Let’s say we have a variable that looks like this:

let s = "hello";

The variable s refers to a string literal, where the value of the string is hardcoded into the text of our program. The variable is valid from the point at which it’s declared until the end of the current scope. Listing 4-1 has comments annotating these places:

{ // s is not valid here, it’s not yet declared

let s = "hello"; // s is valid from this point forward

// do stuff with s

} // this scope is now over, and s is no longer valid

Listing 4-1: A variable and the scope in which it is valid

In other words, there are two important points in time here:

When s comes into scope, it is valid.

It remains so until it goes out of scope.

At this point, the relationship between scopes and when variables are valid is similar to other programming languages. Now we’ll build on top of this understanding by introducing the String type.

The String Type

To illustrate the rules of ownership, we need a data type that is more complex than the ones we covered in Chapter 3. All the data types we’ve looked at previously are stored on the stack and popped off the stack when their scope is over, but we want to look at data that is stored on the heap and explore how Rust knows when to clean up that data.

PROD: Check xref

We’ll use String as the example here and concentrate on the parts of String that relate to ownership. These aspects also apply to other complex data types provided by the standard library and that you create. We’ll discuss String in more depth in Chapter 8.

PROD: check xref

We’ve already seen string literals, where a string value is hardcoded into our program. String literals are convenient, but they aren’t always suitable for every situation in which you want to use text. One reason is that they’re immutable. Another is that not every string value can be known when we write our code: for example, what if we want to take user input and store it? For that situation

, Rust has a second string type, String. This type is allocated on the heap and as such is able to store an amount of text that is unknown to us at compile time. You can create a String from a string literal using the from function, like so:

let s = String::from("hello");

The double colon (::) is an operator that allows us to namespace this particular from function under the String type rather than using some sort of name like string\_from. We’ll discuss this syntax more in the “Method Syntax” section of Chapter 5 and the “Modules” Chapter 7.

PROD: Check chapter xrefs

This kind of string can be mutated:

let mut s = String::from("hello");

s.push\_str(", world!"); // push\_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`

So, what’s the difference here? Why can String be mutated but literals cannot? The difference is how these two types deal with memory.

Memory and Allocation

In the case of a string literal, we know the contents at compile time so the text is hardcoded directly into the final executable, making string literals fast and efficient. But these properties only come from its immutability. Unfortunately, we can’t put a blob of memory into the binary for each piece of text whose size is unknown at compile time and whose size might change while running the program.

With the String type, in order to support a mutable, growable piece of text, we need to allocate an amount of memory on the heap, unknown at compile time, to hold the contents. This means:

The memory must be requested from the operating system at runtime.

We need a way of returning this memory to the operating system when we’re done with our String.

That first part is done by us: when we call String::from, its implementation requests the memory it needs. This is pretty much universal in programming languages.

However, the second part is different. In languages with a garbage collector (GC), the GC keeps track and cleans up memory that isn’t being used anymore, and we, as the programmer, don’t need to think about it. Without a GC, it’s the programmer’s responsibility to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it. Doing this correctly has historically been a difficult programming problem. If we forget, we’ll waste memory. If we do it too early, we’ll have an invalid variable. If we do it twice, that’s a bug too. We need to pair exactly one allocate with exactly one free.

Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope. Here’s a version of our scope example from Listing 4-1 using a String instead of a string literal:

{

let s = String::from("hello"); // s is valid from this point forward

// do stuff with s

} // this scope is now over, and s is no longer valid

There is a natural point at which we can return the memory our String needs to the operating system: when s goes out of scope. When a variable goes out of scope, Rust calls a special function for us. This function is called drop, and it’s where the author of String can put the code to return the memory. Rust calls drop automatically at the closing }.

Note: In C++, this pattern of deallocating resources at the end of an item's lifetime is sometimes called Resource Acquisition Is Initialization (RAII). The drop function in Rust will be familiar to you if you’ve used RAII patterns.

This pattern has a profound impact on the way Rust code is written. It may seem simple right now, but the behavior of code can be unexpected in more complicated situations when we want to have multiple variables use the data we’ve allocated on the heap. Let’s explore some of those situations now.

Ways Variables and Data Interact: Move

Multiple variables can interact with the same data in different ways in Rust. Let’s look at an example using an integer:

let x = 5;

let y = x;

We can probably guess what this is doing based on our experience with other languages: “Bind the value 5 to x; then make a copy of the value in x and bind it to y.” We now have two variables, x and y, and both equal 5. This is indeed what is happening because integers are simple values with a known, fixed size, and these two 5 values are pushed onto the stack.

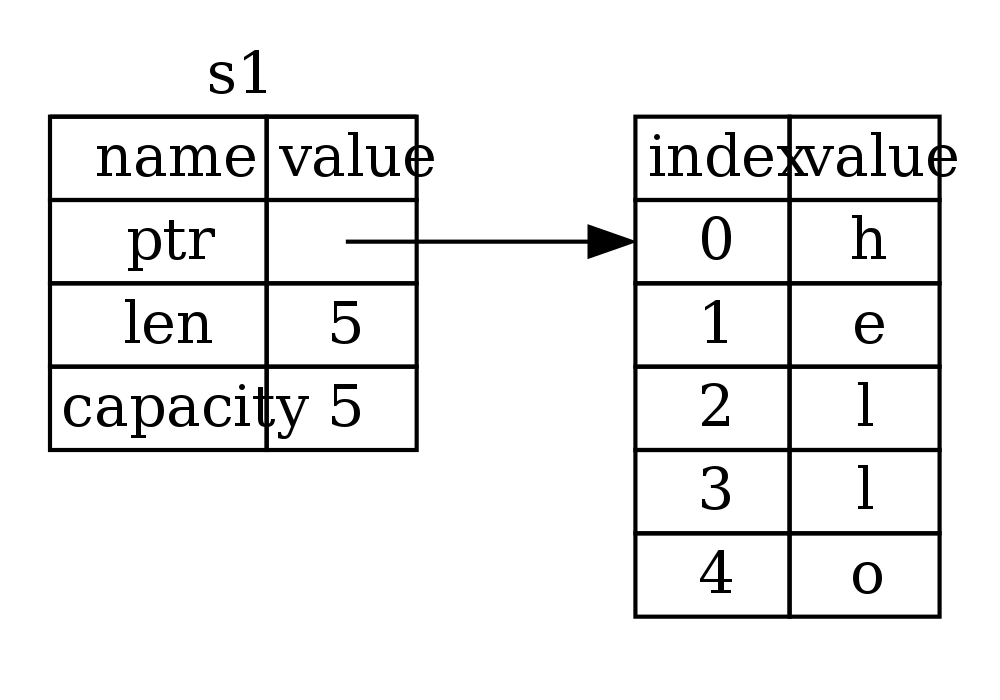
Now let’s look at the String version:

let s1 = String::from("hello");

let s2 = s1;

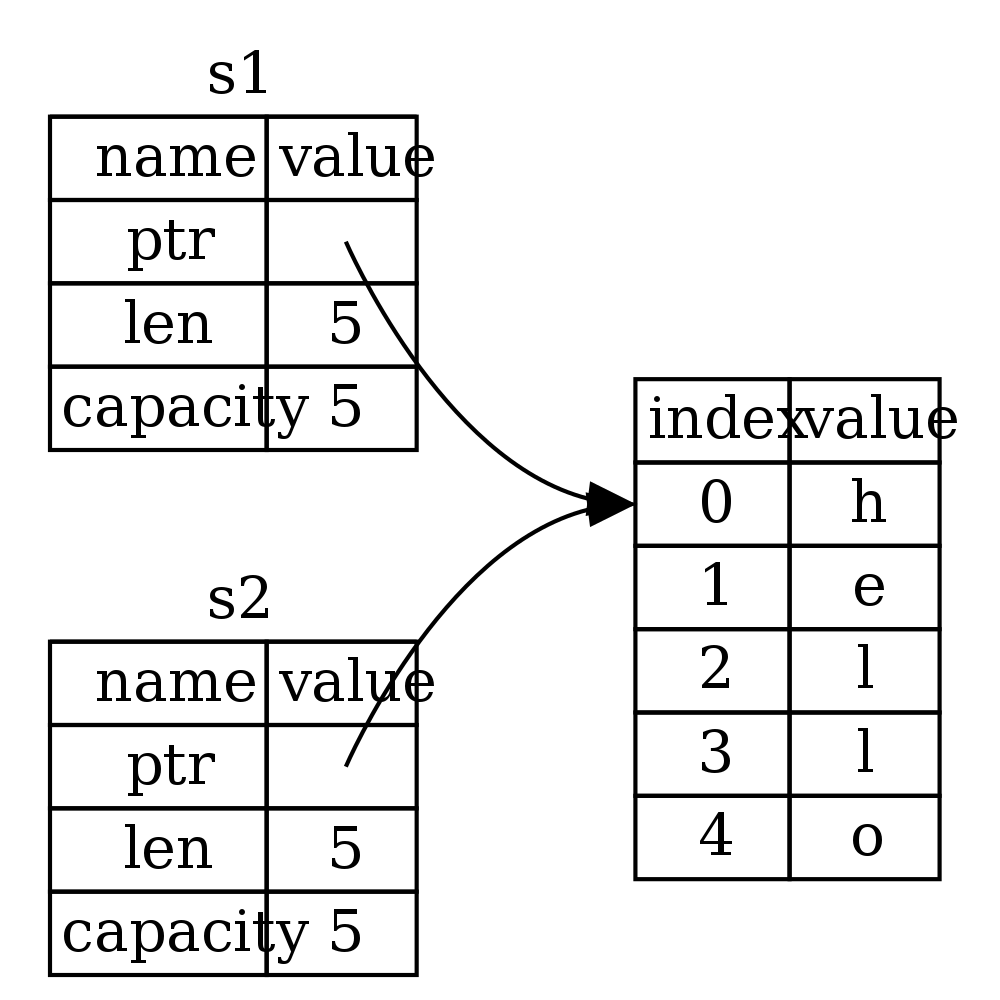
This looks very similar to the previous code, so we might assume that the way it works would be the same: that is, the second line would make a copy of the value in s1 and bind it to s2. But this isn’t quite what happens.

To explain this more thoroughly, let’s look at what String looks like under the covers in Figure 4-2. A String is made up of three parts, shown on the left: a pointer to the memory that holds the contents of the string, a length, and a capacity. This group of data is stored on the stack. On the right is the memory on the heap that holds the contents.

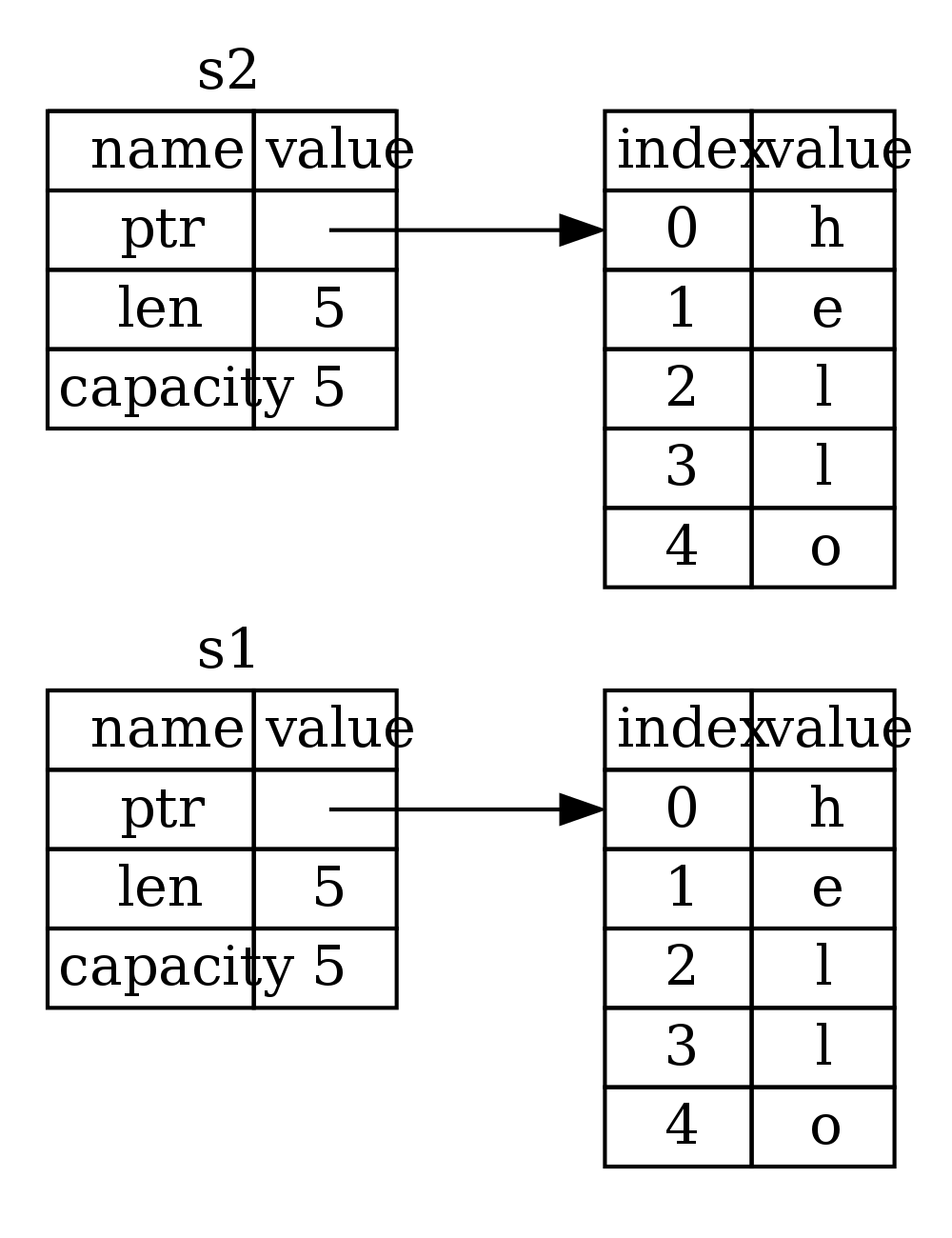
Figure 4-2: Representation in memory of a String holding the value "hello" bound to s1

The length is how much memory, in bytes, the contents of the String is currently using. The capacity is the total amount of memory, in bytes, that the String has received from the operating system. The difference between length and capacity matters, but not in this context, so for now, it’s fine to ignore the capacity.

When we assign s1 to s2, the String data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the String’s pointer refers to. In other words, the data representation in memory looks like Figure 4-3.

Figure 4-3: Representation in memory of the variable s2 that has a copy of the pointer, length, and capacity of s1

The representation does not look like Figure 4-4, which is what memory would look like if Rust instead copied the heap data as well. If Rust did this, the operation s2 = s1 could potentially be very expensive in terms of runtime performance if the data on the heap was large.

Figure 4-4: Another possibility of what s2 = s1 might do if Rust copied the heap data as well

Earlier, we said that when a variable goes out of scope, Rust automatically calls the drop function and cleans up the heap memory for that variable. But Figure 4-3 shows both data pointers pointing to the same location. This is a problem: when s2 and s1 go out of scope, they will both try to free the same memory. This is known as a double free error and is one of the memory safety bugs we mentioned previously. Freeing memory twice can lead to memory corruption, which can potentially lead to security vulnerabilities.

To ensure memory safety, there’s one more detail to what happens in this situation in Rust. Instead of trying to copy the allocated memory, Rust considers s1 to no longer be valid and therefore doesn’t need to free anything when it goes out of scope. Check out what happens when you try to use s1 after s2 is created:

let s1 = String::from("hello");

let s2 = s1;

println!("{}", s1);

You’ll get an error like this because Rust prevents you from using the invalidated reference:

5:22 error: use of moved value: `s1` [E0382]

println!("{}", s1);

^~

5:24 note: in this expansion of println! (defined in <std macros>)

3:11 note: `s1` moved here because it has type `collections::string::String`, which is moved by default

let s2 = s1;

^~

If you’ve heard the terms “shallow copy” and “deep copy” while working with other languages, the concept of copying the pointer, length, and capacity without copying the data probably sounds like a shallow copy. But because Rust also invalidates the first variable, instead of calling this a shallow copy, it’s known as a move. Here we would read this by saying that s1 was movedinto s2. So what actually happens is shown in Figure 4-5.

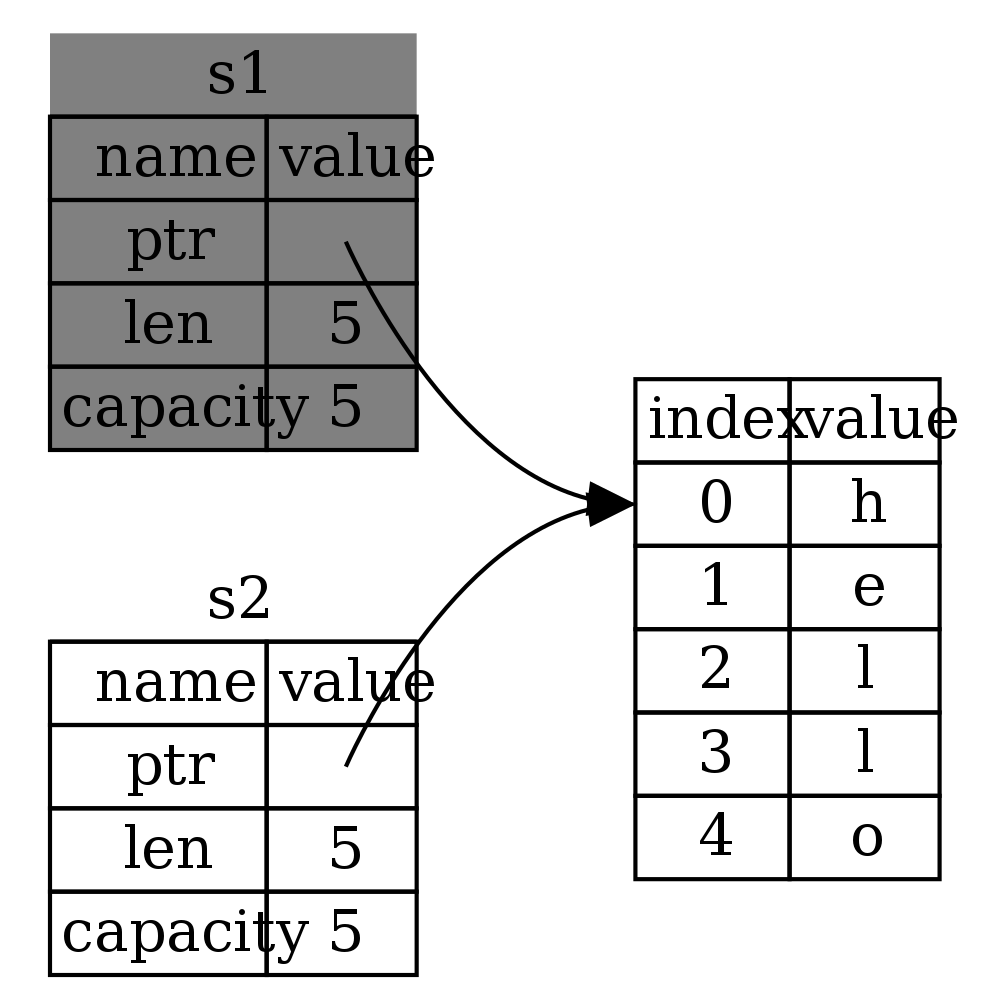


Figure 4-5: Representation in memory after s1 has been invalidated

That solves our problem! With only s2 valid, when it goes out of scope, it alone will free the memory, and we’re done.

In addition, there’s a design choice that’s implied by this: Rust will never automatically create “deep” copies of your data. Therefore, any automaticcopying can be assumed to be inexpensive in terms of runtime performance.

Ways Variables and Data Interact: Clone

If we do want to deeply copy the data in the String, not just the String, we can use a common method called clone. We’ll discuss method syntax in Chapter 5, but because methods are a common feature in many programming languages, you’ve probably seen them before.

PROD: Check xref

Here’s an example of the clone method in action:

let s1 = String::from("hello");

let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);

This works just fine and is how you can explicitly produce the behavior shown in Figure 4-4, where the heap data does get copied.

When you see a call to clone, you know that some arbitrary code is being executed and that code may be expensive. It’s a visual indicator that something different is going on.

Stack-Only Data: Copy

There’s another wrinkle we haven’t talked about yet. This code, shown earlier, works and is valid:

let x = 5;

let y = x;

println!("x = {}, y = {}", x, y);

But this code seems to contradict what we just learned: we don’t have a call to clone, but x is still valid and wasn’t moved into y.

The reason is that types like integers that have a known size at compile time are stored entirely on the stack, so copies of the actual values are quick to make. That means there’s no reason we would want to prevent x from being valid after we create the variable y. In other words, there’s no difference between deep and shallow copying here, so calling clone wouldn’t do anything differently from the usual shallow copying and we can leave it out.

Rust has a special annotation called the Copy trait that we can place on types like integers that are stored on the stack (we’ll talk more about traits in Chapter 10). If a type has the Copy trait, an older variable is still usable after assignment. Rust won’t let us annotate a type with the Copy trait if the type, or any of its parts, has implemented the Drop trait. If the type needs something special to happen when the value goes out of scope and we add the Copy annotation to that type, we’ll get a compile time error.

PROD: Check xref

So what types are Copy? You can check the documentation for the given type to be sure, but as a general rule, any group of simple scalar values can be Copy, and nothing that requires allocation or is some form of resource is Copy. Here are some of the types that are Copy:

All the integer types, like u32.

The boolean type, bool, with values true and false.

All the floating point types, like f64.

Tuples, but only if they contain types that are also Copy. (i32, i32) is Copy, but (i32, String) is not.

Ownership and Functions

The semantics for passing a value to a function are similar to assigning a value to a variable. Passing a variable to a function will move or copy, just like assignment. Listing 4-6 has an example with some annotations showing where variables go into and out of scope:

Filename: src/main.rs

fn main() {

let s = String::from("hello"); // s comes into scope.

takes\_ownership(s); // s's value moves into the function...

// ... and so is no longer valid here.

let x = 5; // x comes into scope.

makes\_copy(x); // x would move into the function,

// but i32 is Copy, so it’s okay to still

// use x afterward.

} // Here, x goes out of scope, then s. But since s's value was moved, nothing

// special happens.

fn takes\_ownership(some\_string: String) { // some\_string comes into scope.

println!("{}", some\_string);

} // Here, some\_string goes out of scope and `drop` is called. The backing

// memory is freed.

fn makes\_copy(some\_integer: i32) { // some\_integer comes into scope.

println!("{}", some\_integer);

} // Here, some\_integer goes out of scope. Nothing special happens.

Listing 4-6: Functions with ownership and scope annotated

If we tried to use s after the call to takes\_ownership, Rust would throw a compile time error. These static checks protect us from mistakes. Try adding code to main that uses s and x to see where you can use them and where the ownership rules prevent you from doing so.

Return Values and Scope

Returning values can also transfer ownership. Here’s an example with similar annotations to those in Listing 4-6:

Filename: src/main.rs

fn main() {

let s1 = gives\_ownership(); // gives\_ownership moves its return

// value into s1.

let s2 = String::from("hello"); // s2 comes into scope.

let s3 = takes\_and\_gives\_back(s2); // s2 is moved into

// takes\_and\_gives\_back, which also

// moves its return value into s3.

} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was

// moved, so nothing happens. s1 goes out of scope and is dropped.

fn gives\_ownership() -> String { // gives\_ownership will move its

// return value into the function

// that calls it.

let some\_string = String::from("hello"); // some\_string comes into scope.

some\_string // some\_string is returned and

// moves out to the calling

// function.

}

// takes\_and\_gives\_back will take a String and return one.

fn takes\_and\_gives\_back(a\_string: String) -> String { // a\_string comes into scope.

a\_string // a\_string is returned and moves out to the calling function.

}

The ownership of variables follows the same pattern every time: assigning a value to another variable moves it, and when heap data values’ variables go out of scope, if the data hasn’t been moved to be owned by another variable, the value will be cleaned up by drop.

Taking ownership and then returning ownership with every function is a bit tedious. What if we want to let a function use a value but not take ownership? It’s quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.

It’s possible to return multiple values using a tuple, like this:

Filename: src/main.rs

fn main() {

let s1 = String::from("hello");

let (s2, len) = calculate\_length(s1);

println!("The length of '{}' is {}.", s2, len);

}

fn calculate\_length(s: String) -> (String, usize) {

let length = s.len(); // len() returns the length of a String.

(s, length)

}

But this is too much ceremony and a lot of work for a concept that should be common. Luckily for us, Rust has a feature for this concept, and it’s called references.

References and Borrowing

The issue with the tuple code at the end of the preceding section is that we have to return the String to the calling function so we can still use the String after the call to calculate\_length, because the String was moved into calculate\_length.

Here is how you would define and use a calculate\_length function that takes a reference to an object as an argument instead of taking ownership of the argument:

Filename: src/main.rs

fn main() {

let s1 = String::from("hello");

let len = calculate\_length(&s1);

println!("The length of '{}' is {}.", s1, len);

}

fn calculate\_length(s: &String) -> usize {

s.len()

}

First, notice that all the tuple code in the variable declaration and the function return value is gone. Second, note that we pass &s1 into calculate\_length, and in its definition, we take &String rather than String.

These & symbols are references, and they allow you to refer to some value without taking ownership of it. Figure 4-7 shows a diagram.

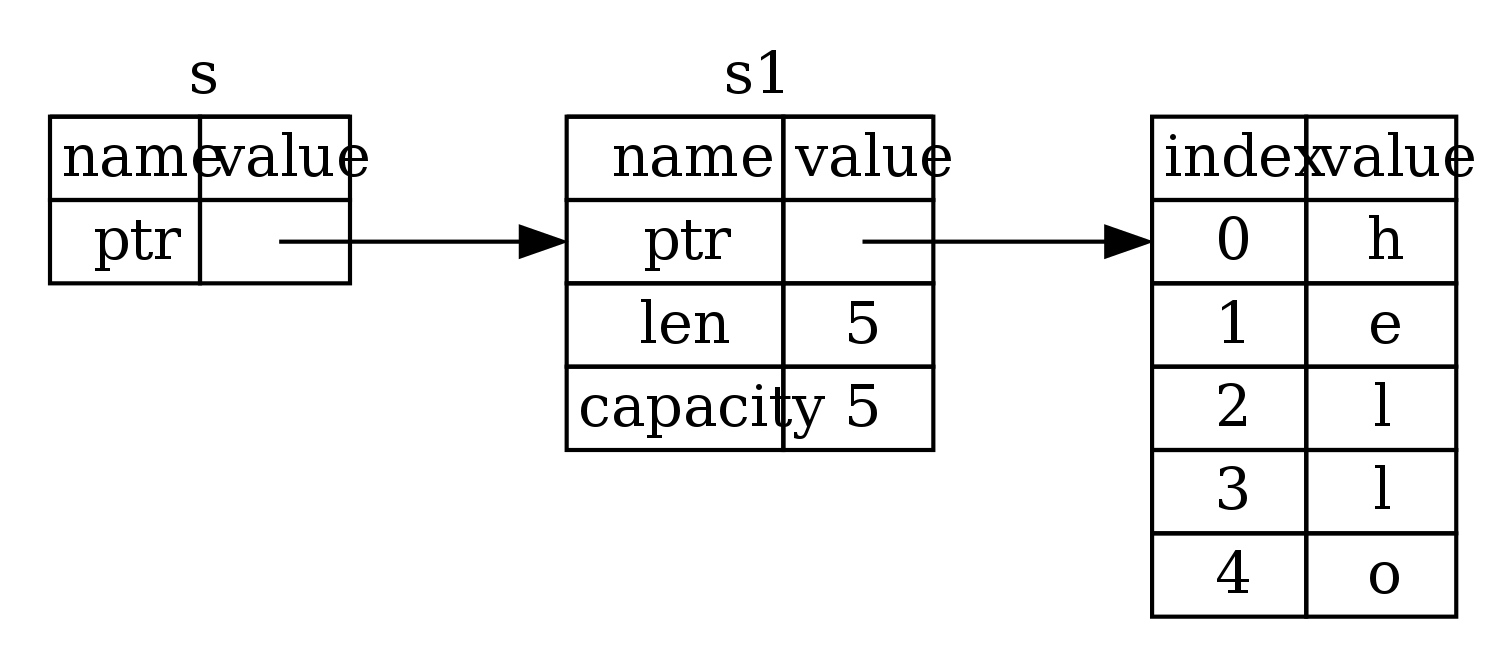


Figure 4-7: &String s pointing at String s1

Let’s take a closer look at the function call here:

let s1 = String::from("hello");

let len = calculate\_length(&s1);

The &s1 syntax lets us create a reference that refers to the value of s1 but does not own it. Because it does not own it, the value it points to will not be dropped when the reference goes out of scope.

Likewise, the signature of the function uses & to indicate that it takes a reference as an argument. Let’s add some explanatory annotations:

fn calculate\_length(s: &String) -> usize { // s is a reference to a String

s.len()

} // Here, s goes out of scope. But because it does not have ownership of what

// it refers to, nothing happens.

The scope in which the variable s is valid is the same as any function argument's scope, but we don’t drop what the reference points to when it goes out of scope because we don’t have ownership. Functions that take references as arguments instead of the actual values mean we won’t need to return the values in order to give back ownership, since we never had ownership.

We call taking references as function arguments borrowing. As in real life, if a person owns something, you can borrow it from them, and when you’re done, you have to give it back.

So what happens if we try to modify something we’re borrowing? Try the code in Listing 4-8. Spoiler alert: it doesn’t work!

Filename: src/main.rs

fn main() {

let s = String::from("hello");

change(&s);

}

fn change(some\_string: &String) {

some\_string.push\_str(", world");

}

Listing 4-8: Attempting to modify a borrowed value

Here’s the error:

error: cannot borrow immutable borrowed content `\*some\_string` as mutable

--> error.rs:8:5

|

8 | some\_string.push\_str(", world");

| ^^^^^^^^^^^

Just as variables are immutable by default, so are references. We’re not allowed to modify something we have a reference to.

Mutable References

We can fix the error in the code from Listing 4-8 with just a small tweak:

Filename: src/main.rs

fn main() {

let mut s = String::from("hello");

change(&mut s);

}

fn change(some\_string: &mut String) {

some\_string.push\_str(", world");

}

First, we had to change s to be mut. Then we had to create a mutable reference with &mut s and accept a mutable reference with some\_string: &mut String.

But mutable references have one big restriction: you can only have one mutable reference to a particular piece of data in a particular scope. This code will fail:

Filename: src/main.rs

let mut s = String::from("hello");

let r1 = &mut s;

let r2 = &mut s;

Here’s the error:

error[E0499]: cannot borrow `s` as mutable more than once at a time

--> borrow\_twice.rs:5:19

|

4 | let r1 = &mut s;

| - first mutable borrow occurs here

5 | let r2 = &mut s;

| ^ second mutable borrow occurs here

6 | }

| - first borrow ends here

This restriction allows for mutation but in a very controlled fashion. It’s something that new Rustaceans struggle with, because most languages let you mutate whenever you’d like. The benefit of having this restriction is that Rust can prevent data races at compile time.

A data race is a particular type of race condition in which these three behaviors occur:

Two or more pointers access the same data at the same time.

At least one of the pointers is being used to write to the data.

There’s no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix when you’re trying to track them down at runtime; Rust prevents this problem from happening because it won’t even compile code with data races!

As always, we can use curly brackets to create a new scope, allowing for multiple mutable references, just not simultaneous ones:

let mut s = String::from("hello");

{

let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with no problems.

let r2 = &mut s;

A similar rule exists for combining mutable and immutable references. This code results in an error:

let mut s = String::from("hello");

let r1 = &s; // no problem

let r2 = &s; // no problem

let r3 = &mut s; // BIG PROBLEM

Here’s the error:

error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable

--> borrow\_thrice.rs:6:19

|

4 | let r1 = &s; // no problem

| - immutable borrow occurs here

5 | let r2 = &s; // no problem

6 | let r3 = &mut s; // BIG PROBLEM

| ^ mutable borrow occurs here

7 | }

| - immutable borrow ends here

Whew! We also cannot have a mutable reference while we have an immutable one. Users of an immutable reference don’t expect the values to suddenly change out from under them! However, multiple immutable references are okay because no one who is just reading the data has the ability to affect anyone else’s reading of the data.

Even though these errors may be frustrating at times, remember that it’s the Rust compiler pointing out a potential bug early (at compile time rather than at runtime) and showing you exactly where the problem is instead of you having to track down why sometimes your data isn’t what you thought it should be.

Dangling References

In languages with pointers, it’s easy to erroneously create a dangling pointer, a pointer that references a location in memory that may have been given to someone else, by freeing some memory while preserving a pointer to that memory. In Rust, by contrast, the compiler guarantees that references will never be dangling references: if we have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

Let’s try to create a dangling reference:

Filename: src/main.rs

fn main() {

let reference\_to\_nothing = dangle();

}

fn dangle() -> &String {

let s = String::from("hello");

&s

}

Here’s the error:

error[E0106]: missing lifetime specifier

--> dangle.rs:5:16

|

5 | fn dangle() -> &String {

| ^^^^^^^

|

= help: this function's return type contains a borrowed value, but there is no

value for it to be borrowed from.

= help: consider giving it a 'static lifetime

error: aborting due to previous error

This error message refers to a feature you haven’t learned about yet: lifetimes. We’ll discuss lifetimes in detail in Chapter 10. But, if you disregard the parts about lifetimes, the message does contain the key to why this code is a problem:

PROD: Check xref

this function's return type contains a borrowed value, but there is no value

for it to be borrowed from.

Let’s take a closer look at exactly what’s happening at each stage of our dangle code:

fn dangle() -> &String { // dangle returns a reference to a String

let s = String::from("hello"); // s is a new String

&s // we return a reference to the String, s

} // Here, s goes out of scope, and is dropped. Its memory goes away.

// Danger!

Because s is created inside dangle, when the code of dangle is finished, it will be deallocated. But we tried to return a reference to it. That means this reference would be pointing to an invalid String! That’s no good. Rust won’t let us do this.

The correct code here is to return the String directly:

fn no\_dangle() -> String {

let s = String::from("hello");

s

}

This works without any problem. Ownership is moved out, and nothing is deallocated.

The Rules of References

Let’s recap what we’ve discussed about references:

At any given time, you can have either but not both of:

One mutable reference.

Any number of immutable references.

References must always be valid.

Next, we’ll look at a different kind of reference: slices.

Slices

Another data type that does not have ownership is slices: slices let you reference a contiguous sequence of elements in a collection rather than the whole collection.

Here’s a small programming problem: write a function that takes a string and returns the first word it finds in that string. If the function doesn’t find a space in the string, it means the whole string is one word, so the entire string should be returned.

Let’s think about the signature of this function:

fn first\_word(s: &String) -> ?

This function, first\_word, takes a &String as an argument. We don’t want ownership, so this is fine. But what should we return? We don’t really have a way to talk about part of a string. However, we could return the index of the end of the word. Let’s try that as shown in Listing 4-9:

Filename: src/main.rs

fn first\_word(s: &String) -> usize {

let bytes = s.as\_bytes();

for (i, &item) in bytes.iter().enumerate() {

if item == b' ' {

return i;

}

}

s.len()

}

Listing 4-9: The first\_word function that returns a byte index value into the String argument

Let’s break down this code a bit:

let bytes = s.as\_bytes();

Because we need to go through the String element by element and check whether a value is a space, we’ll convert our String to an array of bytes using the as\_bytes method:

for (i, &item) in bytes.iter().enumerate() {

We’ll discuss iterators in more detail in Chapter 16. But for now, know that iter is a method that returns each element in a collection, and enumerate wraps the result of iter and returns each element as part of a tuple instead, where the first element of the tuple is the index and the second element is a reference to the element. This is a bit more convenient than calculating the index ourselves.

PROD: Check xref

Because the method uses a tuple, we can use patterns, just like every where else in Rust. So we match against the tuple with i for the index and &item for a single byte. Because we get a reference from .iter().enumerate(), we use & in the pattern:

if item == b' ' {

return i;

}

}

s.len()

We search for the byte that represents the space by using the byte literal syntax. If we find a space, we return the position. Otherwise, we return the length of the string by using s.len().

We now have a way to find out the index of the end of the first word in the string, but there’s a problem. We’re returning a usize on its own, but it’s only a meaningful number in the context of the &String. In other words, because it’s a separate value from the String, there’s no guarantee that it will still be valid in the future. Consider the program in Listing 4-10 that uses the first\_word function from Listing 4-9:

Filename: src/main.rs

fn main() {

let mut s = String::from("hello world");

let word = first\_word(&s); // word will get the value 5.

s.clear(); // This empties the String, making it equal to "".

// word still has the value 5 here, but there's no more string that

// we could meaningfully use the value 5 with. word is now totally invalid!

}

Listing 4-10: Storing the result from calling the first\_word function then changing the String contents

This program compiles without any errors and also would if we used word after calling s.clear(). word isn’t connected to the state of s at all, so word still contains the value 5. We could use that value 5 with the variable s to try to extract the first word out, but this would be a bug because the contents of s have changed since we saved 5 in word.

Having to worry about the index in word getting out of sync with the data in s is tedious and error prone! Managing these indices is even more brittle if we write a second\_word function. Its signature would have to look like this:

fn second\_word(s: &String) -> (usize, usize) {

Now we’re tracking a start and an ending index, and we have even more values that were calculated from data in a particular state but aren’t tied to that state at all. We now have three unrelated variables floating around that need to be kept in sync.

Luckily, Rust has a solution to this problem: string slices.

String Slices

A string slice is a reference to part of a String, and looks like this:

let s = String::from("hello world");

let hello = &s[0..5];

let world = &s[6..11];

This is similar to taking a reference to the whole String but with the extra [0..5] bit. Rather than a reference to the entire String, it’s a reference to an internal position in the String and the number of elements that it refers to.

We create slices with a range of [starting\_index..ending\_index], but the slice data structure actually stores the starting position and the length of the slice. So in the case of let world = &s[6..11];, world would be a slice that contains a pointer to the 6th byte of s and a length value of 5.

Figure 4-11 shows this in a diagram.

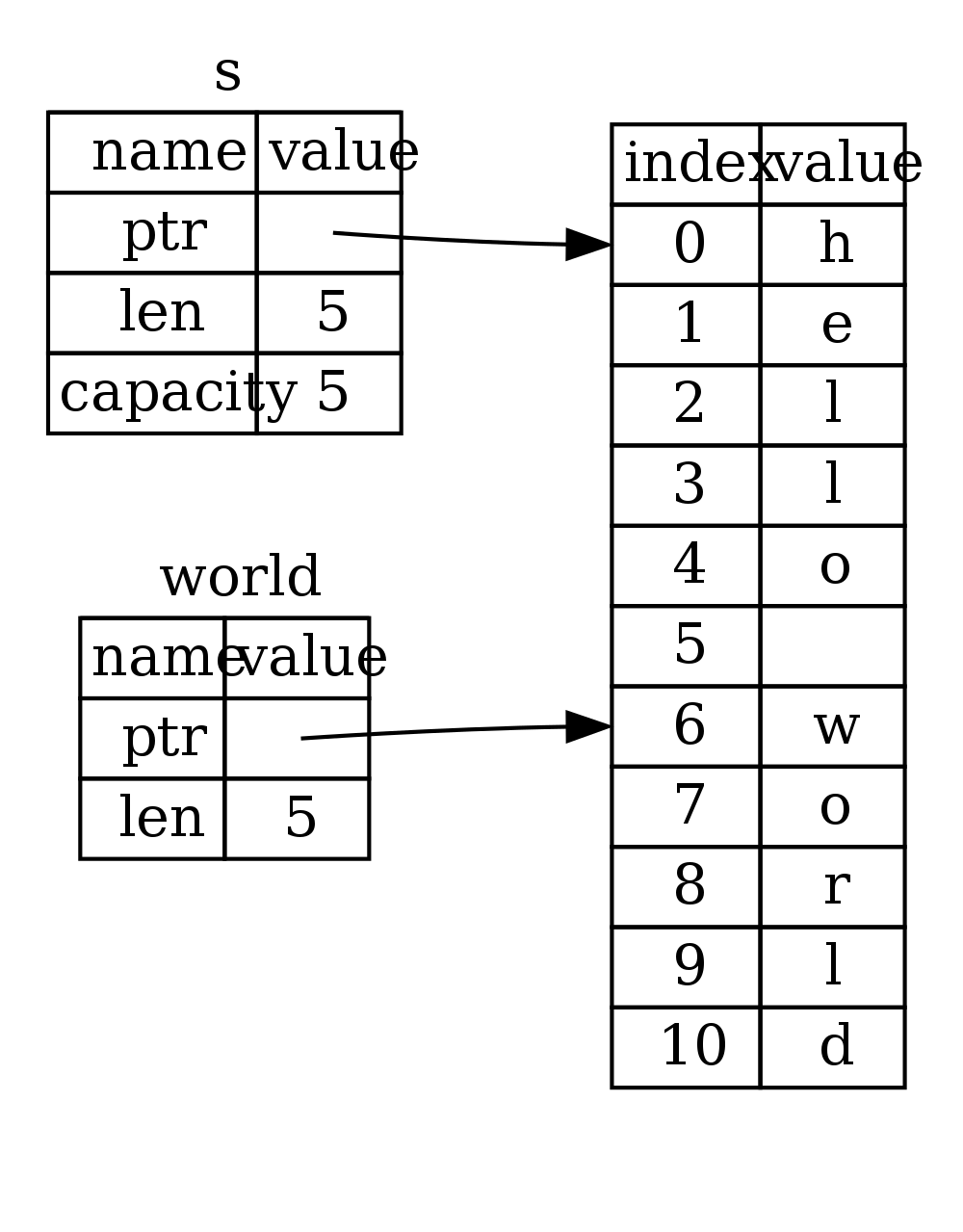


Figure 4-11: String slice referring to part of a String

With Rust’s .. range syntax, if you want to start at the first index (zero), you can drop the value before the ... In other words, these are equal:

let s = String::from("hello");

let slice = &s[0..2];

let slice = &s[..2];

By the same token, if your slice includes the last byte of the String, you can drop the trailing number. That means these are equal:

let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];

let slice = &s[3..];

You can also drop both values to take a slice of the entire string. So these are equal:

let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];

let slice = &s[..];

With all this information in mind, let’s rewrite first\_word to return a slice. The type that signifies “string slice” is written as &str:

Filename: src/main.rs

fn first\_word(s: &String) -> &str {

let bytes = s.as\_bytes();

for (i, &item) in bytes.iter().enumerate() {

if item == b' ' {

return &s[0..i];

}

}

&s[..]

}

We get the index for the end of the word in the same way as we did in Listing 4-9, by looking for the first occurrence of a space. When we find a space, we return a string slice using the start of the string and the index of the space as the starting and ending indices.

Now when we call first\_word, we get back a single value that is tied to the underlying data. The value is made up of a reference to the starting point of the slice and the number of elements in the slice.

Returning a slice would also work for a second\_word function:

fn second\_word(s: &String) -> &str {

We now have a straightforward API that’s much harder to mess up, since the compiler will ensure the references into the String remain valid. Remember the bug in the program in Listing 4-10, when we got the first word but then cleared the string so our first word was invalid? That code was logically incorrect but didn’t show any immediate errors. The problems would show up later if we kept trying to use the first word index with an emptied string. Slices make this bug impossible and let us know we have a problem with our code much sooner. Using the slice version of first\_word will throw a compile time error:

Filename: src/main.rs

fn main() {

let mut s = String::from("hello world");

let word = first\_word(&s);

s.clear(); // Error!

}

Here’s the compiler error:

17:6 error: cannot borrow `s` as mutable because it is also borrowed as

immutable [E0502]

s.clear(); // Error!

^

15:29 note: previous borrow of `s` occurs here; the immutable borrow prevents

subsequent moves or mutable borrows of `s` until the borrow ends

let word = first\_word(&s);

^

18:2 note: previous borrow ends here

fn main() {

}

^

Recall from the borrowing rules that if we have an immutable reference to something, we cannot also take a mutable reference. Because clear needs to truncate the String, it tries to take a mutable reference, which fails. Not only has Rust made our API easier to use, but it has also eliminated an entire class of errors at compile time!

String Literals Are Slices

Recall that we talked about string literals being stored inside the binary. Now that we know about slices, we can properly understand string literals:

let s = "Hello, world!";

The type of s here is &str: it’s a slice pointing to that specific point of the binary. This is also why string literals are immutable; &str is an immutable reference.

String Slices as Arguments

Knowing that you can take slices of literals and Strings leads us to one more improvement on first\_word, and that’s its signature:

fn first\_word(s: &String) -> &str {

A more experienced Rustacean would write the following line instead because it allows us to use the same function on both Strings and &strs:

fn first\_word(s: &str) -> &str {

If we have a string slice, we can pass that as the argument directly. If we have a String, we can pass a slice of the entire String. Defining a function to take a string slice argument instead of a reference to a String makes our API more general and useful without losing any functionality:

Filename: src/main.rs

fn main() {

let my\_string = String::from("hello world");

// first\_word works on slices of `String`s

let word = first\_word(&my\_string[..]);

let my\_string\_literal = "hello world";

// first\_word works on slices of string literals

let word = first\_word(&my\_string\_literal[..]);

// since string literals \*are\* string slices already,

// this works too, without the slice syntax!

let word = first\_word(my\_string\_literal);

}

Other Slices

String slices, as you might imagine, are specific to strings. But there’s a more general slice type, too. Consider this array:

let a = [1, 2, 3, 4, 5];

Just like we might want to refer to a part of a string, we may want to refer to part of an array and would do so like this:

let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

This slice has the type &[i32]. It works the same way as string slices do, by storing a reference to the first element and a length. You’ll use this kind of slice for all sorts of other collections. We’ll discuss these collections in detail when we talk about vectors in Chapter 8.

PROD: Check xref

Summary

The concepts of ownership, borrowing, and slices are what ensure memory safety in Rust programs at compile time. The Rust language gives you control over your memory usage like other systems programming languages, but having the owner of data automatically clean up that data when the owner goes out of scope means you don’t have to write and debug extra code to get this control.

Ownership affects how lots of other parts of Rust work, so we’ll talk about these concepts further throughout the rest of the book. Let’s move on to the next chapter and look at grouping pieces of data together in a struct.